Modular Verification of Recursive Programs

[redacted] [1,2], [redacted] [1,3] and [redacted] [4]

[2] University of Amsterdam, Institute of Language, Logic and Computation, Amsterdam
[3] Leiden Institute of Advanced Computer Science, University of Leiden, The Netherlands
[4] Department of Computing Science, University of Oldenburg, Germany

Abstract. We argue that verification of recursive programs by means of the assertional method of C.A.R. Hoare can be conceptually simplified using a modular reasoning. In this approach some properties of the program are established first and subsequently used to establish other program properties. We illustrate this approach by providing a modular correctness proof of the $Quicksort$ program.

1 Introduction

Program verification by means of the assertional method of Hoare (so-called Hoare's logic) is by now well-understood. One of its drawbacks is that it calls for a tedious manipulation of assertions, which is error prone. The support offered by the available by interactive proof checkers, such as PVS (Prototype Verification System), see [15], is very limited.

One way to reduce the complexity of an assertional correctness proof is by organizing it into a series of simpler proofs. For example, to prove $\{p\}\ S\ \{q_1 \wedge q_2\}$ we could establish $\{p\}\ S\ \{q_1\}$ and $\{p\}\ S\ \{q_2\}$ separately (and *independently*). Such an obvious approach is clearly of very limited use.

In this paper we propose a different approach that is appropriate for recursive programs. In this approach a simpler property, say $\{p_1\}\ S\ \{q_1\}$, is established first and then *used in the proof* of another property, say $\{p_2\}\ S\ \{q_2\}$. This allows us to establish $\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}$ in a modular way. It is obvious how to generalize this approach to an arbitrary sequence of program properties for which the earlier properties are used in the proofs of the latter ones. So, in contrast to the simplistic approach mentioned above, the proofs of the program properties are *not* independent but are arranged instead into an acyclic directed graph.

We illustrate this approach by providing a modular correctness proof of the $Quicksort$ program due to [10]. This yields a correctness proof that is better structured and conceptually easier to understand than the original one, given in [7]. A minor point is that we use different proof rules concerning procedure calls and also provide an assertional proof of termination of the program, a property not considered in [7]. (It should be noted that termination of recursive procedures with parameters within the framework of the assertional method was considered only in the eighties, see, e.g., [2]. In these proofs some subtleties arise that necessitate a careful exposition, see [1].)

We should mention here two other references concerning formal verification of the $Quicksort$ program. In [6] the proof of $Quicksort$ is certified using the interactive

O. Grumberg et al. (Eds.): Francez Festschrift, LNCS 5533, pp. 1–21, 2009.

theorem prover Coq, while in [13] a correctness proof of a non-recursive version of $Quicksort$ is given.

The paper is organized as follows. In the next section we introduce a small programming language that involves recursive procedures with parameters called by value and discuss its operational semantics. Then, in Section 3 we introduce a proof system for proving partial and total correctness of these programs. The presentation in these two sections is pretty standard except for the treatment of the call-by-value parameter mechanism that avoids the use of substitution.

Next, in Section 4 we discuss how the correctness proofs, both of partial and of total correctness, can be structured in a modular way. In Section 5 we illustrate this approach by proving correctness of the $Quicksort$ program while in Section 6 we discuss related work and draw some conclusions. Finally, in the appendix we list the used axioms and proof rules concerned with non-recursive programs. The soundness of the considered proof systems is rigorously established in [3] using the operational semantics of [16,17].

2 A Small Programming Language

Syntax

We use *simple* variables and *array* variables. Simple variables are of a basic type (for example **integer** or **Boolean**), while array variables are of a higher type (for example **integer** $\times$ **Boolean** $\rightarrow$ **integer**). A *subscripted variable* derived from an array variable a of type $T_1 \times \ldots \times T_n \rightarrow T$ is an expression of the form $a[t_1, \ldots, t_n]$, where each expression t_i is of type T_i.

In this section we introduce a class of recursive programs as an extension of the class of **while** programs which are generated by the following grammar:

$$S ::= skip \mid u := t \mid \bar{x} := \bar{t} \mid S_1;\ S_2 \mid \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } B \textbf{ do } S_1 \textbf{ od},$$

where S stands for a typical statement or program, u for a simple or subscripted variable, t for an expression (of the same type as u), and B for a Boolean expression. Further, $\bar{x} := \bar{t}$ is a parallel assignment, with $\bar{x} = x_1, \ldots, x_n$ a non-empty list of distinct simple variables and $\bar{t} = t_1, \ldots, t_n$ a list of expressions of the corresponding types. The parallel assignment plays a crucial role in our modelling of the parameter passing. We do not discuss the types and only assume that the set of basic types includes at least the types **integer** and **Boolean**. As an abbreviation we introduce **if** B **then** S **fi** $\equiv$ **if** B **then** S **else** $skip$ **fi**.

Given an expression t, we denote by $var(t)$ the set of all simple and array variables that appear in t. Analogously, given a program S, we denote by $var(S)$ the set of all simple and array variables that appear in S, and by $change(S)$ the set of all simple and array variables that can be modified by S, i.e., the set of variables that appear on the left-hand side of an assignment in S.

We arrive at recursive programs by adding recursive procedures with call-by-value parameters. To distinguish between local and global variables, we first introduce a *block statement* by the grammar rule

$$S ::= \textbf{begin local } \bar{x} := \bar{t}; S_1 \textbf{ end}.$$

A block statement introduces a non-empty sequence $\bar{x}$ of simple local variables, all of which are explicitly initialized by means of a parallel assignment $\bar{x} := \bar{t}$, and provides an explicit scope for these simple local variables. The precise explanation of a scope is more complicated because the block statements can be nested.

Assuming $\bar{x} = x_1, \ldots, x_k$ and $\bar{t} = t_1, \ldots, t_k$, each occurrence of a local variable x_i within the statement S_1 *and not* within another block statement that is a subprogram of S_1 refers to the same variable. Each such variable x_i is initialized to the expression t_i by means of the parallel assignment $\bar{x} := \bar{t}$. Further, given a statement S' such that **begin local** $\bar{x} := \bar{t}; S_1$ **end** is a subprogram of S', all occurrences of x_i in S' outside this block statement refer to some other variable(s).

Procedure calls with parameters are introduced by the grammar rule

$$S ::= P(t_1, \ldots, t_n),$$

where P is a procedure identifier and $t_1, \ldots, t_n$, with $n \geq 0$, are expressions called *actual parameters*. The statement $P(t_1, \ldots, t_n)$ is called a *procedure call*. The resulting class of programs is then called *recursive programs*.

Procedures are defined by *declarations* of the form

$$P(u_1, \ldots, u_n) :: S,$$

where $u_1, \ldots, u_n$ are distinct simple variables, called *formal parameters* of the procedure P and S is the *body* of the procedure P.

We assume a given set of procedure declarations D such that each procedure that appears in D has a unique declaration in D. When considering recursive programs we assume that all procedures whose calls appear in the considered recursive programs are declared in D. Additionally, we assume that the procedure calls are *well-typed*, which means that the numbers of formal and actual parameters agree and that for each parameter position the types of the corresponding actual and formal parameters coincide.

Given a recursive program S, we call a variable x_i *local* if it appears within a subprogram of D or S of the form **begin local** $\bar{x} := \bar{t}; S_1$ **end** with $\bar{x} = x_1, \ldots, x_k$, and *global* otherwise.

To avoid possible name clashes between local and global variables we assume that given a set of procedure declarations D and a recursive program S, no local variable of S occurs in D. So given the procedure declaration

$$P :: \textbf{if } x = 1 \textbf{ then } b := \textbf{true else } b := \textbf{false fi}$$

the program

$$S \equiv \textbf{begin local } x := 1; P \textbf{ end}$$

is not allowed. If it were, the semantics we are about to introduce would allow us to conclude that $\{x = 0\}\ S\ \{b\}$ holds. However, the customary semantics of the programs in the presence of procedures prescribes that in this case $\{x = 0\}\ S\ \{\neg b\}$ should hold, as the meaning of a program should not depend on the choice of the names of its local variables. (This is a consequence of the so-called *static scope* of the variables that we assume here.)

This problem is trivially solved by just renaming the 'offensive' local variables to avoid name clashes, so by considering here the program **begin local** $y := 1; P$ **end** instead of S. Once we limit ourselves to recursive programs no local variable of which occurs in the considered set of procedure declarations, the semantics we introduce ensures that the names of local variables indeed do not matter. More precisely, the programs that only differ in the choice of the names of local variables and obey the above syntactic restriction have then identical meaning. In what follows, when considering a recursive program S in the context of a set of procedure declarations D we always implicitly assume that the above syntactic restriction is satisfied.

The local and global variables play an analogous role to the bound and free variables in first-order formulas or in λ-terms. In fact, the above syntactic restriction corresponds to the 'Variable Convention' of [4, page 26] according to which "all bound variables are chosen to be different from the free variables."

Note that the above definition of programs puts no restrictions on the actual parameters in procedure calls; in particular they can be formal parameters or global variables.

Semantics

For recursive programs we use a structural operational semantics in the sense of Plotkin [17]. As usual, it is defined in terms of transitions between configurations. A *configuration* C is a pair $< S,\ \sigma >$ consisting a statement S that is to be executed and a state σ that assigns a value to each variable (including local variables). A *transition* is written as a step $C \to C'$ between configurations. To express termination we use the empty statement E; a configuration $< E,\ \sigma >$ denotes termination in the state σ.

Transitions are specified by the transition axioms and rules which are defined in the context of a set D of procedure declarations. The only transition axioms that are somewhat non-standard are the ones that deal with the block statement and the procedure calls, in that they avoid the use of substitution thanks to the use of parallel assignment:

$< \textbf{begin local } \bar{x} := \bar{t}; S \textbf{ end}, \sigma > \ \to \ < \bar{x} := \bar{t}; S; \bar{x} := \sigma(\bar{x}), \sigma >,$

$< P(\bar{t}); R, \sigma > \ \to \ < \textbf{begin local } \bar{u} := \bar{t}; S \textbf{ end}; R, \sigma >,$

where $P(\bar{u}) :: S \in D$.

The first axiom ensures that the local variables are initialized as prescribed by the parallel assignment and that upon termination the global variables whose names coincide with the local variables are restored to their initial values, held at the beginning of the block statement. This is a way of implicitly modeling a *stack discipline* for (nested) blocks. So the use of the block statement in the second transition axiom ensures that prior to the execution of the procedure body the formal parameters are *simultaneously* instantiated to the actual parameters and that upon termination of a procedure call the formal parameters are restored to their initial values. Additionally, the block statement limits the scope of the formal parameters so that they are not accessible upon termination of the procedure call. So the second transition axiom describes the *call-by-value* parameter mechanism.

Based on the transition relation $\rightarrow$ we consider two variants of input/output semantics for recursive programs S refering to the set Σ of states σ, τ. The *partial correctness semantics* is a mapping $\mathcal{M}[\![S]\!] : \Sigma \rightarrow \mathcal{P}(\Sigma)$ defined by

$$\mathcal{M}[\![S]\!](\sigma) = \{\tau \mid < S, \sigma > \ \rightarrow^* \ < E, \tau >\}.$$

The *total correctness semantics* is a mapping $\mathcal{M}_{tot}[\![S]\!] : \Sigma \rightarrow \mathcal{P}(\Sigma \cup \{\bot\})$ defined by

$$\mathcal{M}_{tot}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) \cup \{\bot \mid S \text{ can diverge from } \sigma\}.$$

Here $\bot$ is an error state signalling divergence, i.e., an infinite sequence of transitions starting in the configuration $< S, \sigma >$.

3 Proof Systems for Partial and Total Correctness

Program correctness is expressed by *correctness formulas* of the form $\{p\}\ S\ \{q\}$, where S is a program and p and q are assertions. The assertion p is the *precondition* of the correctness formula and q is the *postcondition*. A correctness formula $\{p\}\ S\ \{q\}$ is true in the sense of partial correctness if every terminating computation of S that starts in a state satisfying p terminates in a state satisfying q. And $\{p\}\ S\ \{q\}$ is true in the sense of total correctness if every computation of S that starts in a state satisfying p terminates and its final state satisfies q. Thus in the case of partial correctness, diverging computations of S are not taken into account.

Using the semantics $\mathcal{M}$ and $\mathcal{M}_{tot}$, we formalize these two interpretations of correctness formulas uniformly as set theoretic inclusions as follows (cf. [3]). For an assertion p let $[\![p]\!]$ denote the set of states satisfying p. Then we define:

(i) The correctness formula $\{p\}\ S\ \{q\}$ is true in the sense of *partial correctness*, abbreviated by $\models \{p\}\ S\ \{q\}$, if $\mathcal{M}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!]$.

(ii) The correctness formula $\{p\}\ S\ \{q\}$ is true in the sense of *total correctness*, abbreviated by $\models_{tot} \{p\}\ S\ \{q\}$, if $\mathcal{M}_{tot}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!]$.

Since by definition $\bot \notin [\![q]\!]$, part (ii) indeed formalizes the above intuition about total correctness.

Partial Correctness

Partial correctness of **while** programs is proven using the customary proof system *PD* consisting of the group of axioms and rules 1–7 shown in the appendix. Consider now partial correctness of recursive programs. First, we introduce the following rule that deals with the block statement.

BLOCK

$$\frac{\{p\}\ \bar{x} := \bar{t}; S\ \{q\}}{\{p\}\ \textbf{begin local}\ \bar{x} := \bar{t}; S\ \textbf{end}\ \{q\}}$$

where $var(\bar{x}) \cap free(q) = \emptyset$.

By $free(q)$ we denote here the set of all free simple and array variables that have a free occurrence in the assertion q.

The main issue is how to deal with the procedure calls. To this end, we want to adjust the proofs of 'generic' procedure calls to arbitrary ones. The definition of a generic call and the conditions for the correctness of such an adjustment process refer to the assumed set of procedure declarations D. By a generic call of a procedure P we mean a call of the form $P(\bar{x})$, where $\bar{x}$ is a sequence of fresh (w.r.t. D) variables.

First, we extend the definition of $change(S)$ to recursive programs and sets of procedure declarations as follows:

$$
\begin{array}{l}
change(\textbf{begin local } \bar{x} := \bar{t}; S \textbf{ end}) = change(S) \setminus \{\bar{x}\},\\
change(P(\bar{u}) :: S) = change(S) \setminus \{\bar{u}\},\\
change(\{P(\bar{u}) :: S\} \cup D) = change(P(\bar{u}) :: S) \cup change(D),\\
change(P(\bar{t})) = \emptyset.
\end{array}
$$

The adjustment of the generic procedure calls is then taken care of by the following proof rule that refers to the set of procedure declarations D:

INSTANTIATION

$$\frac{\{p\}\ P(\bar{x})\ \{q\}}{\{p[\bar{x} := \bar{t}]\}\ P(\bar{t})\ \{q[\bar{x} := \bar{t}]\}}$$

where $var(\bar{x}) \cap var(D) = var(\bar{t}) \cap change(D) = \emptyset$ and $P(\bar{u}) :: S \in D$ for some S.

In the following rule for recursive procedures with parameters we use the provability symbol $\vdash$ to refer to the proof system *PD* augmented with the auxiliary axiom and rules A1–A6 defined in the appendix and the above two proof rules.

RECURSION

$$\frac{\begin{array}{l}\{p_1\}\ P_1(\bar{x}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{x}_n)\ \{q_n\} \vdash \{p\}\ S\ \{q\},\\ \{p_1\}\ P_1(\bar{x}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{x}_n)\ \{q_n\} \vdash\\ \qquad \{p_i\}\ \textbf{begin local } \bar{u}_i := \bar{x}_i; S_i \textbf{ end}\ \{q_i\},\ i \in \{1, \ldots, n\}\end{array}}{\{p\}\ S\ \{q\}}$$

where $D = P_1(\bar{u}_1) :: S_1, \ldots, P_n(\bar{u}_n) :: S_n$ and $var(\bar{x}_i) \cap var(D) = \emptyset$ for $i \in \{1, \ldots, n\}$.

The intuition behind this rule is as follows. Say that a program S is (p, q)-*correct* if $\{p\}\ S\ \{q\}$ holds in the sense of partial correctness. The second premise of the rule states that we can establish from the *assumption* of the (p_i, q_i)-correctness of the 'generic' procedure calls $P_i(\bar{x}_i)$ for $i \in \{1, \ldots, n\}$, the (p_i, q_i)-correctness of the procedure bodies S_i for $i \in \{1, \ldots, n\}$, which are adjusted as in the transition axiom that deals with the procedure calls. Then we can prove the (p_i, q_i)-correctness of the procedure calls $P_i(\bar{x}_i)$ unconditionally, and thanks to the first premise establish the (p, q)-correctness of the recursive program S.

To prove *p*artial correctness of *r*ecursive programs with parameters we use the proof system *PR* that is obtained by extending the proof system *PD* by the block rule, the instantiation rule, the recursion rule, and the auxiliary axiom and rules A1–A6.

Note that when we deal only with one recursive procedure and use the procedure call as the considered recursive program, this rule simplifies to

$$\frac{\{p\}\ P(\bar{x})\ \{q\} \vdash \{p\}\ \textbf{begin local}\ \bar{u} := \bar{x}; S\ \textbf{end}\ \{q\}}{\{p\}\ P(\bar{x})\ \{q\}}$$

where $D = P(\bar{u}) :: S$ and $var(\bar{x}) \cap var(D) = \emptyset$.

Total Correctness

Total correctness of **while** programs is proven using the proof system *TD* consisting of the group of axioms and rules 1–5, 7, and 8 shown in the appendix. For total correctness of recursive programs we need a modification of the recursion rule. The provability symbol $\vdash$ refers now to the proof system *TD* augmented with the auxiliary rules A2–A6, the block rule and the instantiation rule. The proof rule is a minor variation of a rule originally proposed in [1] and has the following form:

RECURSION II

$$\frac{\begin{array}{l}\{p_1\}\ P_1(\bar{x}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{x}_n)\ \{q_n\} \vdash \{p\}\ S\ \{q\},\\ \{p_1 \wedge t < z\}\ P_1(\bar{x}_1)\ \{q_1\}, \ldots, \{p_n \wedge t < z\}\ P_n(\bar{x}_n)\ \{q_n\} \vdash\\ \quad \{p_i \wedge t = z\}\ \textbf{begin local}\ \bar{u}_i := \bar{x}_i; S_i\ \textbf{end}\ \{q_i\},\ i \in \{1, \ldots, n\}\end{array}}{\{p\}\ S\ \{q\}}$$

where $D = P_1(\bar{u}_1) :: S_1, \ldots, P_n(\bar{u}_n) :: S_n$, $var(\bar{x}_i) \cap var(D) = \emptyset$ for $i \in \{1, \ldots, n\}$, and z is an integer variable that does not occur in p_i, t, q_i and S_i for $i \in \{1, \ldots, n\}$ and is treated in the proofs as a constant, which means that in these proofs neither the $\exists$-introduction rule A4 nor the substitution rule A6 defined in the appendix is applied to z.

To prove *t*otal correctness of *r*ecursive programs with parameters we use the proof system *TR* that is obtained by extending the proof system *TD* by the block rule, the instantiation rule, the recursion rule II, and the auxiliary rules A2–A6.

As before, in the case of one recursive procedure this rule can be simplified to

$$\frac{\begin{array}{l}\{p \wedge t < z\}\ P(\bar{x})\ \{q\} \vdash \{p \wedge t = z\}\ \textbf{begin local}\ \bar{u} := \bar{x}; S\ \textbf{end}\ \{q\},\\ p \rightarrow t \geq 0\end{array}}{\{p\}\ P(\bar{x})\ \{q\}}$$

where $D = P(\bar{u}) :: S$, $var(\bar{x}) \cap var(D) = \emptyset$ and z is an integer variable that does not occur in p, t, q and S and is treated in the proof as a constant.

4 Modularity

Proof system *TR* allows us to establish total correctness of recursive programs directly. However, sometimes it is more convenient to decompose the proof of total correctness into two separate proofs, one of partial correctness and one of termination. More

specifically, given a correctness formula $\{p\}\ S\ \{q\}$, we first establish its partial correctness, using proof system *PR*. Then, to show termination it suffices to prove the simpler correctness formula $\{p\}\ S\ \{\mathbf{true}\}$ using proof system *TR*.

These two different proofs can be combined into one using the following general proof rule for total correctness:

DECOMPOSITION

$$\frac{\vdash_{PR} \{p\}\ S\ \{q\},\quad \vdash_{TR} \{p\}\ S\ \{\mathbf{true}\}}{\{p\}\ S\ \{q\}}$$

where $\vdash_{PR}$ and $\vdash_{PR}$ refer to the proofs in the proof systems *PR* and *TR*, respectively.

The decomposition rule and other auxiliary rules like A2 or A3 allow us to combine two correctness formulas derived *independently*. In some situations it is helpful to reason about procedure calls in a hierarchical way, by first deriving one correctness formula and then using it in a proof of another correctness formula. The following modification of the above simplified version of the recursion rule illustrates this principle, where we limit ourselves to a two-stage proof and one procedure:

MODULARITY

$$\frac{\{p_0\}\ P(\bar{x})\ \{q_0\} \vdash \{p_0\}\ \mathbf{begin\ local}\ \bar{u} := \bar{x}; S\ \mathbf{end}\ \{q_0\},\quad \{p_0\}\ P(\bar{x})\ \{q_0\}, \{p\}\ P(\bar{x})\ \{q\} \vdash \{p\}\ \mathbf{begin\ local}\ \bar{u} := \bar{x}; S\ \mathbf{end}\ \{q\}}{\{p\}\ P(\bar{x})\ \{q\}}$$

where $D = P(\bar{u}) :: S$ and $var(\bar{x}) \cap var(D) = \emptyset$.

So first we derive an auxiliary property, $\{p_0\}\ P(\bar{x})\ \{q_0\}$ that we subsequently use in the proof of the 'main' property, $\{p\}\ P(\bar{x})\ \{q\}$. In general, more procedures may be used and an arbitrary 'chain' of auxiliary properties may be constructed. In the next section we show that such a modular approach can lead to better structured correctness proofs.

5 Correctness Proof of the $Quicksort$ Procedure

We now apply the modular proof method to verify total correctness of the *Quicksort* algorithm, originally introduced in [10]. For a given array a of type $\mathbf{integer} \to \mathbf{integer}$ and integers x and y this algorithm sorts the section $a[x : y]$ consisting of all elements $a[i]$ with $x \leq i \leq y$. Sorting is accomplished 'in situ', i.e., the elements of the initial (unsorted) array section are permuted to achieve the sorting property. We consider here the following version of *Quicksort* close to the one studied in [7]. It consists of a recursive procedure $Quicksort(m, n)$, where the formal parameters m, n and the local variables v, w are all of type $\mathbf{integer}$:

$Quicksort(m,n)$ ::
  if $m < n$
  then $Partition(m,n)$;
    begin
    local $v, w := ri, le$;
    $Quicksort(m,v)$;
    $Quicksort(w,n)$
    end
  fi

$Quicksort$ calls a non-recursive procedure $Partition(m,n)$ which partitions the array a suitably, using global variables ri, le, pi of type **integer** standing for *pivot*, *left*, and *right* elements:

$Partition(m,n)$ ::
  $pi := a[m]$;
  $le, ri := m, n$;
  while $le \leq ri$ **do**
    while $a[le] < pi$ **do** $le := le + 1$ **od**;
    while $pi < a[ri]$ **do** $ri := ri - 1$ **od**;
    if $le \leq ri$ **then**
      $swap(a[le], a[ri])$;
      $le, ri := le + 1, ri - 1$
    fi
  od

Here for two given simple or subscripted variables u and v the program $swap(u,v)$ is used to *swap* the values of u and v. So we stipulate that the following correctness formula

$$\{x = u \wedge y = v\}\ swap(u,v)\ \{x = v \wedge y = u\}$$

holds in the sense of partial and total correctness, where x and y are fresh variables.

In the following D denotes the set of the above two procedure declarations and S_Q the body of the procedure $Quicksort(m,n)$.

Formal Problem Specification

Correctness of $Quicksort$ amounts to proving that upon termination of the procedure call $Quicksort(m,n)$ the array section $a[m:n]$ is sorted and is a permutation of the input section. To write the desired correctness formula we introduce some notation. The assertion

$$sorted(a[x:y]) \equiv \forall i,j : (x \leq i \leq j \leq y \rightarrow a[i] \leq a[j])$$

states that the integer array section $a[x:y]$ is sorted. To express the permutation property we use an auxiliary array a_0 in the section $a_0[x:y]$ of which we record the initial values of $a[x:y]$. The abbreviation

$$bij(f,x,y) \ \equiv\ f \text{ is a bijection on } \mathbb{Z} \ \wedge\ \forall\, i \notin [x:y] : f(i) = i$$

states that f is a bijection on $\mathbb{Z}$ which is the identity outside the interval $[x:y]$. Hence

$$perm(a, a_0, [x : y]) \equiv \exists f : (bij(f, x, y) \wedge \forall i : a[i] = a_0[f(i)])$$

specifies that the array section $a[x : y]$ is a permutation of the array section $a_0[x : y]$ and that a and a_0 are the same elsewhere.

We can now express the correctness of $Quicksort$ by means of the following correctness formula:

Q1 $\{a = a_0\}\ Quicksort(x, y)\ \{perm(a, a_0, [x : y]) \wedge sorted(a[x : y])\}$.

To prove correctness of $Quicksort$ in the sense of partial correctness we proceed in stages and follow the methodology explained in Section 4. In other words, we establish some auxiliary correctness formulas first, using among others the recursion rule. Then we use them as premises in order to derive other correctness formulas, also using the recursion rule.

Properties of $Partition$

In the proofs we shall use a number of properties of the $Partition$ procedure. This procedure is non-recursive, so to verify them it suffices to prove the corresponding properties of the procedure body using the proof systems *PD* and *TD*, a task we leave to Nissim Francez.

More precisely, we assume the following properties of $Partition$ in the sense of partial correctness:

P1 $\{\mathbf{true}\}\ Partition(m, n)\ \{ri \leq n \wedge m \leq le\}$,

P2 $\{x' \leq m \wedge n \leq y' \ \wedge\ perm(a, a_0, [x' : y'])\}$
$Partition(m, n)$
$\{x' \leq m \wedge n \leq y' \ \wedge\ perm(a, a_0, [x' : y'])\}$,

P3 $\{\mathbf{true}\}$
$Partition(m, n)$
$\{\ le > ri \wedge$
$(\forall i \in [m : ri] :\ a[i] \leq pi) \wedge$
$(\forall i \in [ri + 1 : le - 1] :\ a[i] = pi) \wedge$
$(\forall i \in [le : n] :\ pi \leq a[i])\}$,

and the following property in the sense of total correctness:

P4 $\{m < n\}$
$Partition(m, n)$
$\{ri - m < n - m \wedge n - le < n - m\}$.

Property **P1** states the bounds for ri and le. We remark that $le \leq n$ and $m \leq ri$ need not hold upon termination. Property **P2** implies that the call $Partition(n, k)$ permutes the array section $a[m : n]$ and leaves other elements of a intact, but actually is a stronger

statement involving an interval $[x' : y']$ that includes $[m : n]$, so that we can carry out the reasoning about the recursive calls of $Quicksort$. Finally, property **P3** captures the main effect of the call $Partition(n, k)$: the elements of the section $a[m : n]$ are rearranged into three parts, those smaller than pi (namely $a[m : ri]$), those equal to pi (namely $a[ri + 1 : le - 1]$), and those larger than pi (namely $a[le : n]$). Property **P4** is needed in the termination proof of the $Quicksort$ procedure: it states that the subsections $a[m : ri]$ and $a[le : n]$ are strictly smaller than the section $a[m : n]$.

Auxiliary proof: permutation property

In the remainder of this section we use the following abbreviation:

$$J \equiv m = x \wedge n = y.$$

We first extend the permutation property **P2** to the procedure $Quicksort$:

$$\begin{array}{ll} \textbf{Q2} & \{perm(a, a_0, [x' : y']) \wedge x' \leq x \wedge y \leq y'\} \\ & Quicksort(x, y) \\ & \{perm(a, a_0, [x' : y'])\} \end{array}$$

Until further notice the provability symbol $\vdash$ refers to the proof system *PD* augmented with the the block rule, the instantiation rule and the auxiliary rules A2–A6.

The appropriate claim needed for the application of the recursion rule is:

Claim 1

$$\begin{array}{l} \textbf{P1}, \textbf{P2}, \textbf{Q2} \vdash \{perm(a, a_0, [x' : y']) \wedge x' \leq x < y \leq y'\} \\ \qquad \textbf{begin local } m, n := x, y; S_Q \textbf{ end} \\ \qquad \{perm(a, a_0, [x' : y'])\}. \end{array}$$

Proof. In Figure 1 a proof outline is given that uses as assumptions the correctness formulas **P1**, **P2**, and **Q2**. More specifically, the used correctness formula about the call of $Partition$ is derived from **P1** and **P2** by the conjunction rule. In turn, the correctness formulas about the recursive calls of $Quicksort$ are derived from **Q2** by an application of the instantiation rule and the invariance rule. This concludes the proof of Claim 1. □

We can now derive **Q2** by the recursion rule. In summary, we proved

$$\textbf{P1},\ \textbf{P2} \vdash \textbf{Q2}.$$

Auxiliary proof: sorting property

We can now verify that the call $Quicksort(x, y)$ sorts the array section $a[x : y]$, so

Q3 $\{\textbf{true}\}\ Quicksort(x, y)\ \{sorted(a[x : y])\}$.

The appropriate claim needed for the application of the recursion rule is:

Claim 2

$$\textbf{P3},\ \textbf{Q2},\ \textbf{Q3} \vdash \{\textbf{true}\}\ \textbf{begin local } m, n := x, y; S_Q \textbf{ end}\ \{sorted(a[x : y])\}.$$

$\{perm(a, a_0, [x' : y']) \wedge x' \leq x \wedge y \leq y'\}$
begin local
$\{perm(a, a_0, [x' : y']) \wedge x' \leq x \wedge y \leq y'\}$
$m, n := x, y;$
$\{perm(a, a_0, [x' : y']) \wedge x' \leq x \wedge y \leq y' \wedge J\}$
$\{perm(a, a_0, [x' : y']) \wedge x' \leq m \ \wedge n \leq y'\}$
if $m < n$ **then**
$\{perm(a, a_0, [x' : y']) \wedge x' \leq m \ \wedge n \leq y'\}$
$Partition(m, n);$
$\{perm(a, a_0, [x' : y']) \wedge x' \leq m \wedge n \leq y' \wedge ri \leq n \wedge m \leq le\}$
begin local
$\{perm(a, a_0, [x' : y']) \wedge x' \leq m \wedge n \leq y' \wedge ri \leq n \wedge m \leq le\}$
$v, w := ri, le;$
$\{perm(a, a_0, [x' : y']) \wedge x' \leq m \wedge n \leq y' \wedge v \leq n \wedge m \leq w\}$
$\{perm(a, a_0, [x' : y']) \wedge x' \leq m \wedge v \leq y' \wedge x' \leq w \wedge n \leq y'\}$
$Quicksort(m, v);$
$\{perm(a, a_0, [x' : y']) \wedge x' \leq w \wedge n \leq y'\}$
$Quicksort(w, n)$
$\{perm(a, a_0, [x' : y'])\}$
end
$\{perm(a, a_0, [x' : y'])\}$
fi
$\{perm(a, a_0, [x' : y'])\}$
end
$\{perm(a, a_0, [x' : y'])\}$

Fig. 1. Proof outline showing permutation property **Q2**

Proof. In Figure 2 a proof outline is given that uses as assumptions the correctness formulas **P3**, **Q2**, and **Q3**. In the following we justify the correctness formulas about $Partition$ and the recursive calls of $Quicksort$ used in this proof outline. In the post-condition of $Partition$ we use the following abbreviation:

$$\begin{aligned} K \equiv \ & v < w \wedge \\ & (\forall\, i \in [m : v] :\ a[i] \leq pi) \wedge \\ & (\forall\, i \in [v + 1 : w - 1] :\ a[i] = pi) \wedge \\ & (\forall\, i \in [w : n] :\ pi \leq a[i]). \end{aligned}$$

Observe that the correctness formula

$$\{J\}\ Partition(m, n)\ \{J \wedge K[v, w := ri, le]\}$$

is derived from **P3** by the invariance rule. Next we verify the correctness formulas

$$\{J \wedge K\} Quicksort(m, v) \{sorted(a[m : v]) \wedge J \wedge K\}, \tag{1}$$

$\{\mathbf{true}\}$
begin local
$\{\mathbf{true}\}$
$m, n := x, y;$
$\{J\}$
if $m < n$ **then**
  $\{J \wedge m < n\}$
  $Partition(m, n);$
  $\{J \wedge K[v, w := ri, le]\}$
  begin local
  $\{J \wedge K[v, w := ri, le]\}$
  $v, w := ri, le;$
  $\{J \wedge K\}$
  $Quicksort(m, v);$
  $\{sorted(a[m : v]) \wedge J \wedge K\}$
  $Quicksort(w, n)$
  $\{sorted(a[m : v] \wedge sorted(a[w : n] \wedge J \wedge K\}$
  $\{sorted(a[x : v] \wedge sorted(a[w : y] \wedge K[m, n := x, y]\}$
  $\{sorted(a[x : y])\}$
  end
  $\{sorted(a[x : y])\}$
fi
$\{sorted(a[x : y])\}$
end
$\{sorted(a[x : y])\}$

Fig. 2. Proof outline showing sorting property **Q3**

and

$$\begin{array}{l}\{sorted(a[m : v]) \wedge J \wedge K\}\\ Quicksort(w, n)\\ \{sorted(a[m : v] \wedge sorted(a[w : n] \wedge J \wedge K\}.\end{array} \tag{2}$$

about the recursive calls of $Quicksort$.

Proof of (1). By applying the instantiation rule to **Q3**, we obtain

A1 $\{\mathbf{true}\}\ Quicksort(m, v)\ \{sorted(a[m : v])\}$.

Moreover, by the invariance axiom, we have

A2 $\{J\}\ Quicksort(m, v)\ \{J\}$.

By applying the instantiation rule to **Q2**, we then obtain

$$\begin{array}{l}\{perm(a, a_0, [x' : y']) \wedge x' \leq m \wedge v \leq y'\}\\ Quicksort(m, v)\\ \{perm(a, a_0, [x' : y'])\}.\end{array}$$

Applying next the substitution rule with the substitution $[x', y' := m, v]$ yields

$$\{perm(a, a_0, [m : v]) \wedge m \leq m \wedge v \leq v\}$$
$$Quicksort(m, v)$$
$$\{perm(a, a_0, [m : v])\}.$$

So by a trivial application of the consequence rule, we obtain

$$\{a = a_0\}\ Quicksort(m, v)\ \{perm(a, a_0, [m : v])\}.$$

We then obtain by an application of the invariance rule

$$\{a = a_0 \wedge K[a := a_0]\}\ Quicksort(m, v)\ \{perm(a, a_0, [m : v]) \wedge K[a := a_0]\}.$$

Note now the following implications:

$$K \rightarrow \exists a_0 : (a = a_0 \wedge K[a := a_0]),$$
$$perm(a, a_0, [m : v]) \wedge K[a := a_0] \rightarrow K.$$

So we conclude

A3 $\{K\}\ Quicksort(m, v)\ \{K\}$

by the $\exists$-introduction rule and the consequence rule. Combining the correctness formulas **A1**–**A3** by the conjunction rule we get (1).

Proof of (2). In a similar way as above, we can prove the correctness formula

$$\{a = a_0\}\ Quicksort(w, n)\ \{perm(a, a_0, [w : n])\}.$$

By an application of the invariance rule we obtain

$$\{a = a_0 \wedge sorted(a_0[m : v]) \wedge v < w\}$$
$$Quicksort(w, n)$$
$$\{perm(a, a_0, [w : n]) \wedge sorted(a_0[m : v]) \wedge v < w\}.$$

Note now the following implications:

$$v < w \wedge sorted(a[m : v]) \rightarrow \exists a_0 : (a = a_0 \wedge sorted(a_0[m : v]) \wedge v < w),$$
$$(perm(a, a_0, [w : n]) \wedge sorted(a_0[m : v]) \wedge v < w) \rightarrow sorted(a[m : v]).$$

So we conclude

B1 $\{v < w \wedge sorted(a[m : v])\}\ Quicksort(w, n)\ \{sorted(a[m : v])\}$

by the $\exists$-introduction rule and the consequence rule. Further, by applying the instantiation rule to **Q3** we obtain

B2 $\{\mathbf{true}\}\ Quicksort(w, n)\ \{sorted(a[w : n])\}$.

Next, by the invariance axiom we obtain

B3 $\{J\}\ Quicksort(w, m)\ \{J\}$.

Further, using the implications

$$K \to \exists a_0 : (a = a_0 \wedge K[a := a_0]),$$
$$perm(a, a_0, [w : n]) \wedge K[a := a_0] \to K,$$

we can derive from **Q2**, in a similar manner as in the proof of **A3**,

B4 $\{K\}\ Quicksort(w, n)\ \{K\}$.

Combining the correctness formulas **B1**–**B4** by the conjunction rule and observing that $K \to v < w$ holds, we get (2).

The final application of the consequence rule in the proof outline given in Figure 2 is justified by the following crucial implication:

$$sorted(a[x : v]) \wedge sorted(a[w : y]) \wedge K[m, n := x, y] \to$$
$$sorted(a[x : y]).$$

Also note that $J \wedge m \geq n \to sorted(a[x : y])$, so the implicit **else** branch is properly taken care of. This concludes the proof of Claim 2. □

We can now derive **Q3** by the recursion rule. In summary, we proved

$$\textbf{P3},\ \textbf{Q2} \vdash \textbf{Q3}.$$

The proof of partial correctness of $Quicksort$ is now immediate: it suffices to combine **Q2** and **Q3** by the conjunction rule. Then after applying the substitution rule with the substitution $[x', y' := x, y]$ and the consequence rule we obtain **Q1**, or more precisely

$$\textbf{P1},\ \textbf{P2},\ \textbf{P3} \vdash \textbf{Q1}.$$

Total Correctness

To prove termination, by the decomposition rule discussed in Section 4, it suffices to establish

Q4 $\{\textbf{true}\}\ Quicksort(x, y)\ \{\textbf{true}\}$

in the sense of total correctness. In the proof we rely on the property **P4** of $Partition$:

$$\{m < n\}\ Partition(m, n)\ \{ri - m < n - m \wedge n - le < n - m\}.$$

The provability symbol $\vdash$ refers below to the proof system *TD* augmented with the block rule, the instantiation rule and the the auxiliary rules A2–A6. For the termination proof of the recursive procedure call $Quicksort(x, y)$ we use

$$t \equiv \max(y - x, 0)$$

as the bound function. Since $t \geq 0$ holds, the appropriate claim needed for the application of the recursion rule II is:

Claim 3

$$\begin{array}{l}\textbf{P4},\ \{t < z\}\ Quicksort(x, y)\ \{\textbf{true}\} \vdash \\ \{t = z\}\ \textbf{begin local}\ m, n := x, y; S_Q\ \textbf{end}\ \{\textbf{true}\}.\end{array}$$

Proof. In Figure 3 a proof outline for total correctness is given that uses as assumptions the correctness formulas **P4** and $\{t < z\}\ Quicksort(x, y)\ \{\textbf{true}\}$. In the following we

$\{t = z\}$
begin local
$\{\max(y - x, 0) = z\}$
$m, n := x, y;$
$\{\max(n - m, 0) = z\}$
if $n < k$ **then**
  $\{\max(n - m, 0) = z \wedge m < n\}$
  $\{n - m = z \wedge m < n\}$
  $Partition(m, n);$
  $\{n - m = z \wedge m < n \wedge ri - m < n - m \wedge n - le < n - m\}$
  begin local
  $\{n - m = z \wedge m < n \wedge ri - m < n - m \wedge n - le < n - m\}$
  $v, w := ri, le;$
  $\{n - m = z \wedge m < n \wedge v - m < n - m \wedge n - w < n - m\}$
  $\{\max(v - m, 0) < z \wedge\ \max(n - w, 0) < z\}$
  $Quicksort(m, v);$
  $\{\max(n - w, 0) < z\}$
  $Quicksort(w, n)$
  $\{\textbf{true}\}$
  end
  $\{\textbf{true}\}$
fi
$\{\textbf{true}\}$
end
$\{\textbf{true}\}$

Fig. 3. Proof outline establishing termination of the $Quicksort$ procedure

justify the correctness formulas about $Partition$ and the recursive calls of $Quicksort$ used in this proof outline. Since $m, n, z \notin change(D)$, we deduce from **P4** using the invariance rule the correctness formula

$$\begin{array}{l}\{n - m = z \wedge m < n\} \\ Partition(m, n) \\ \{n - m = z \wedge ri - m < n - m \wedge n - le < n - m\}.\end{array} \tag{3}$$

Consider now the assumption

$$\{t < z\}\ Quicksort(x, y)\ \{\mathbf{true}\}.$$

Since $n, w, z \notin change(D)$, the instantiation rule and the invariance rule yield

$$\begin{array}{l}\{\max(v - m, 0) < z \wedge\ \max(n - w, 0) < z\}\\ Quicksort(m, v)\\ \{\max(n - w, 0) < z\}\end{array}$$

and

$$\{\max(n - w, 0) < z\}\ Quicksort(w, n)\ \{\mathbf{true}\}.$$

The application of the consequence rule preceding the first recursive call of $Quicksort$ is justified by the following two implications:

$$(n - m = z \wedge m < n \wedge v - m < n - m) \rightarrow \max(v - m, 0) < z,$$
$$(n - m = z \wedge m < n \wedge n - w < n - m) \rightarrow \max(n - w, 0) < z.$$

This completes the proof of Claim 3. □

Applying now the simplified version of the recursion rule II we derive **Q4**. In summary, we proved

$$\mathbf{P4} \vdash \mathbf{Q4}.$$

6 Conclusions

The issue of modularity has been by now well-understood in the area of program construction. It also has been addressed in the program verification. Let us just mention two references, an early one and a recent one: [8] focused on modular verification of temporal properties of concurrent programs which were modelled as a set of modules that interact by means of procedure calls. In turn, [19] considered modular verification of heap manipulating programs, where the focus has been on the automatic extraction and verification specifications.

However, to our knowledge no approach has been proposed to deal with correctness of recursive programs in a modular fashion. When proving correctness of the $Quicksort$ program we found that the simple approach here proposed allowed us to structure the proof better by establishing the 'permutation property' first and then using it in the proof of the 'sorting property'.

So in our approach we propose modularity at the level of *proofs* and not at the level of *programs*. This should be of help when organizing a mechanically verified correctness proof, by expressing the proofs of the subsidiary properties as subsidiary lemmas. In general, modular correctness proofs of programs are proofs from assumptions about subprograms, which can be considered as 'black boxes' of the given programs. Zwiers [20] has investigated an appropriate notion of completeness for such proofs from assumptions about black boxes, called *modular completeness*.

The first proof of partial correctness of *Quicksort* is given in [7]. That proof establishes the permutation and the sorting property simultaneously, in contrast to our

approach. For dealing with recursive procedures, [7] use proof rules corresponding to our rules for blocks, instantiation, and recursion (for the case of one recursive procedure). They also use a so-called *adaptation rule* of [11] that allows one to adapt a given correctness formula about a program to other pre- and postconditions. In our approach we use several auxiliary rules which together have the same effect as the adaptation rule. The expressive power of the adaptation rule has been analyzed in [14]. No proof rule for the termination of recursive procedures is proposed in [7], only an informal argument is given why *Quicksort* terminates. An informal proof of total correctness of *Partition* is given in [12] as part of the program *Find* given in [9].

The recursion rule is modelled after the so-called Scott induction rule for fixed points that appeared first in the unpublished manuscript Scott and De Bakker [18]. Recursion rule II for total correctness is taken from America and De Boer [1], where also the completeness of a proof system similar to *TR* is established. The modularity rule corresponds to a theorem due to Bekić [5] which states that for systems of monotonic functions iterative fixed points coincide with simultaneous fixed points.

Acknowledgment

We thank the reviewer for helpful suggestions.

References

1. America, P., de Boer, F.S.: Proving total correctness of recursive procedures. Information and Computation 84(2), 129–162 (1990)
2. Apt, K.R.: Ten years of Hoare's logic, a survey, part I. ACM Transactions on Programming Languages and Systems 3, 431–483 (1981)
3. Apt, K.R., de Boer, F.S., Olderog, E.-R.: Verification of Sequential and Concurrent Programs, 3rd extended edn. Springer, New York (2009) (to appear)
4. Barendregt, H.P.: The Lambda Calculus. North Holland, Amsterdam (1984)
5. Bekić, H.: Definable operations in general algebras, and the theory of automata and flow charts. Technical report, IBM Laboratory, Vienna (1969); Typescript
6. Filliâtre, J.-C., Magaud, N.: Certification of sorting algorithms in the system Coq. In: Theorem Proving in Higher Order Logics: Emerging Trends (1999)
7. Foley, M., Hoare, C.A.R.: Proof of a recursive program: Quicksort. Computer Journal 14(4), 391–395 (1971)
8. Hailpern, B., Owicki, S.: Modular verification of concurrent programs. In: POPL 1982: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 322–336. ACM, New York (1982)
9. Hoare, C.A.R.: Algorithm 65, Find. Communications of the ACM 4(7), 321 (1961)
10. Hoare, C.A.R.: Quicksort. Comput. J. 5(1), 10–15 (1962)
11. Hoare, C.A.R.: Procedures and parameters: an axiomatic approach. In: Engeler, E. (ed.) Proceedings of Symposium on the Semantics of Algorithmic Languages, New York. Lecture Notes in Mathematics, vol. 188, pp. 102–116. Springer, Heidelberg (1971)
12. Hoare, C.A.R.: Proof of a program: Find. Communications of the ACM 14(1), 39–45 (1971)
13. Kaldewaij, A.: Programming: The Derivation of Algorithms. Prentice-Hall, Englewood Cliffs (1990)

14. Olderog, E.-R.: On the notion of expressiveness and the rule of adaptation. Theoretical Computer Science 30, 337–347 (1983)
15. Owre, S., Shankar, N.: Writing PVS proof strategies. In: Archer, M., Di Vito, B., Muñoz, C. (eds.) Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003), number CP-2003-212448 in NASA Conference Publication, Hampton, VA, September 2003, pp. 1–15. NASA Langley Research Center (2003)
16. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI-FN 19, Department of Computer Science, Aarhus University (1981)
17. Plotkin, G.D.: A structural approach to operational semantics. J. of Logic and Algebraic Programming, 60–61, 17–139 (2004); Revised version of [16]
18. Scott, D., de Bakker, J.W.: A theory of programs. Notes of an IBM Vienna Seminar (1969)
19. Taghdiri, M.: Automating Modular Program Verification by Refining Specifications. Ph.D thesis. MIT, Cambridge, Mass (2008), `http://alloy.mit.edu/community/files/mana_thesis.pdf`
20. Zwiers, J.: Compositionality, Concurrency, and Partial Correctness. LNCS, vol. 321. Springer, Heidelberg (1989)

Appendix

We list here the used axioms and proof rules that were not defined earlier in the text. To establish correctness of **while** programs we rely on the following axioms and proof rules. In the proofs of partial correctness the loop rule is used, while in the proofs of total correctness the loop II rule is used.

AXIOM 1: SKIP

$$\{p\}\ skip\ \{p\}$$

AXIOM 2: ASSIGNMENT

$$\{p[u := t]\}\ u := t\ \{p\}$$

AXIOM 3: PARALLEL ASSIGNMENT

$$\{p[\bar{x} := \bar{t}]\}\ \bar{x} := \bar{t}\ \{p\}$$

RULE 4: COMPOSITION

$$\frac{\{p\}\ S_1\ \{r\}, \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$$

RULE 5: CONDITIONAL

$$\frac{\{p \wedge B\}\ S_1\ \{q\}, \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}$$

RULE 6: LOOP

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \neg B\}}$$

RULE 7: CONSEQUENCE

$$\frac{p \rightarrow p_1, \{p_1\}\ S\ \{q_1\}, q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

RULE 8: LOOP II

$$\frac{\begin{array}{l}\{p \wedge B\}\ S\ \{p\},\\ \{p \wedge B \wedge t = z\}\ S\ \{t < z\},\\ p\ \rightarrow t \geq 0\end{array}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \neg\ B\}}$$

where t is an integer expression and z is an integer variable that does not appear in p, B, t or S.

Additionally, we rely on the following auxiliary axioms and proof rules that occasionally refer to the assumed set of procedure declarations D.

AXIOM A1: INVARIANCE

$$\{p\}\ S\ \{p\}$$

where $free(p) \cap (change(D) \cup change(S)) = \emptyset$.

RULE A2: DISJUNCTION

$$\frac{\{p\}\ S\ \{q\}, \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

RULE A3: CONJUNCTION

$$\frac{\{p_1\}\ S\ \{q_1\}, \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

RULE A4: $\exists$-INTRODUCTION

$$\frac{\{p\}\ S\ \{q\}}{\{\exists x : p\}\ S\ \{q\}}$$

where $x \notin change(D) \cup change(S) \cup free(q)$.

RULE A5: INVARIANCE

$$\frac{\{r\}\ S\ \{q\}}{\{p \wedge r\}\ S\ \{p \wedge q\}}$$

where $free(p) \cap (change(D) \cup change(S)) = \emptyset$.

RULE A6: SUBSTITUTION

$$\frac{\{p\}\ S\ \{q\}}{\{p[\bar{z} := \bar{t}]\}\ S\ \{q[\bar{z} := \bar{t}]\}}$$

where $(var(\bar{z}) \cup var(\bar{t})) \cap (change(D) \cup change(S)) = \emptyset$.